Predicting web application crashes using machine learning

DADS-121

ABSTRACT
Unplanned system outages have a negative impact on company revenues and image. While the last decades have seen a lot of efforts from industry and academia to avoid them, they still happen and their impact is increasing. According to many studies, one of the most important causes of these outages is resource exhaustion for different reasons: overload, inadequate system resource planning, or transient software errors which consume resources until crash. Several previous work have proposed the use of machine learning algorithms for modeling and predicting resource consumption, and the effectiveness of these approaches have been demonstrated in failureless, stationary circumstances. In this paper, we present a comparison of machine learning (ML) techniques to predict the time to crash when the system suffers transient software errors which consume resources randomly and gradually. Furthermore, we present briefly a framework based on these ML techniques to help to avoid downtime, if it is possible. The experiments illustrate that our approach is effective at predicting crashes and with a lot of potential impact.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Reliability, availability and serviceability; D.4.8 [Operating Systems]: Performance—Modeling and prediction

Keywords
Dependability, Machine Learning, TPC-W, Web applications

1. INTRODUCTION
Enterprise environments are rapidly changing, as new needs appear. In particular, availability of the information all the time and from everywhere is today a common requirement. To achieve these new challenges demanded by the industry and society, new IT infrastructures have had to be created. Applications have to interact among themselves and with the environment to achieve these new goals, resulting in complex IT infrastructures that need brilliant IT professionals with hard-to-obtain skills to manage them. However, the complexity is achieving such a level that even the best administrators can hardly cope with it, and only autonomic computing seems to be the solution[1].

Because system complexity is growing day by day, the number of failures due (directly or indirectly) to this complexity has also been growing, resulting in undesirable behaviors, poor levels of service, or even total outages. The need to prevent or gracefully deal with outages of business and critical systems is clear considering attention of the huge loss due the downtime per hour for the industry, as reported in [2][3]. Moreover, outages have a negative impact on the company image that could affect indirectly profits. For this reason, building high availability servers has become a hot topic in the last years.

Clustering [4] is possibly the most used technique to minimize outages in industry, and today most business-critical servers apply some sort of server-redundancy, load-balancers, and fail-over techniques. This is certainly a solution to handle application crashes, but has the inconvenience of additional hardware cost. A more recent trend complements redundancy with self-healing techniques [1], which help automate recovery procedures and prevent the occurrence of unplanned failures. Also, virtualization technologies are been used by the industry to deploy clustering techniques reducing the total cost of ownership (TCO) of the systems [5].

The literature divides the reasons for downtime in three main categories: Human or operator errors, software errors, and hardware errors. According to [9], the distribution of errors in these categories was (approximately) 40%, 40% and 20%, respectively, by 2005. If we observe in detail the evolution of this distribution from 1985 to 2005 [6, 7, 8, 9], we can see that the proportion of hardware errors has decreased (from 32% in the 80s to 15-20% by now). The number of operator errors remains fairly stable along time: Although system complexity has grown and keeps growing, and this would suggest an increase in human errors during the management tasks, the operators currently have better administration tools to automate some parts of their task. Nevertheless this 40% shows clearly that there is important room for improvement. On the other hand, the software errors have been growing up along the time (from 25% in 80s to 40% today), due mainly to the complexity of the current software and the heterogeneous environment where our systems have to work.
According to [6, 7, 8, 9], the most complex failures to overcome are which are due to transient or intermittent errors. These transient or intermittent errors are caused by dormant faults that, under certain unknown circumstances, wake up and temporary disappear after a while. The transient/interruption errors which cause exhaustion of some resource (memory, CPU time, processes, connectivity, etc.) are the most complex errors to predict or model due their "randomly" behavior.

To be precise about causes, effects, and observable effects, we use the terms understand fault, error/bug, and failure following the definitions in [10]. Briefly, a fault is the cause of an error; it is active when it causes an error, otherwise it is a dormant fault. An error or bug is a part of the total state of the system that may lead to its subsequent service failure. And a failure is a visible erroneous operation of the system, probably due to an error or a set of them.

Resource exhaustion could be caused by different reasons, and some of them cannot be considered due to faults or errors. For example, system overload: the system works correctly, but it does not have enough resources to deal with the workload it receives. In our opinion, could be understood like anomaly but not a software error. On the other hand, transient software errors are more difficult to deal with, because they often depend on unforeseen interactions among components, appear under unusual logical conditions, and are hard to reproduce in isolation so that developers can fix them. But they are important: if transient errors happen keep consuming resources (such as memory or CPU) they invariably make the system unstable and lead to performance degradation, visible failures or even crash. We have to take into account that transient software errors are due to permanent faults, however these permanent faults are transient in nature, because sometimes they are dormant.

Machine learning algorithms have been used to predict system behavior under normal circumstances (no errors) to estimate resource usage under varying workloads, and for capacity planning. There are, however, fewer studies in using them to indeterministic degradation and crashes prediction. The power of machine learning approach in front of simple thresholds to predict the resource consumption is due to its capability to adapt itself under dynamic/changeable circumstances.

1.1 Our contribution

In the preceding paper [**], we presented a preliminary study of which prediction methods were adequate for predicting the time until crash under a random error which consumes memory, and concluded that methods combining decision trees and linear regressions seemed to give much better results than either decision trees or linear regression alone. Now we take those conclusions for granted, but here we: 1) present our underlying mathematical model and assumptions in detail, 2) present a more detailed evaluation of the algorithms under new software errors, 3) present an experimental setup that can test performance in dynamically varying the resource injection rate (as opposed to testing under a single workload with fixed resource variation injection), and 4) concentrate on the problem of detecting approaching and imminent crashes (“orange alerts” and “red alerts”) rather than trying to produce accurate time-to-failure estimates long before failures happen. The alarm-detecting is the crucial element to apply the self-healing or demand the human operators attention.

The rest of the paper is organized as follows: Section 2 presents the related work in the area. Section 3 presents in detail our mathematical model and our prediction strategy. Section 4 describes the details of our framework. Section 5 describes the experimental environment used to execute the experiments and the results obtained, and Section 6 presents some conclusion and future work.

2. RELATED WORK

The idea of predicting when a system will exhaust a given resource is not new. A lot of works have modeled this behavior using different machine learning and analytic approaches with successful results. Works such as [11, 12] present different techniques to predict resource exhaustion due to a workload in a system that suffers software aging. In these two works they present two different approaches: in [11], authors use a semi-Markov reward model using the workload and resource usage data collected from the system to predict resource exhaustion in time. In [12], authors use time-series ARMA models from the system data to estimate the resource exhaustion due to workload received by the system.

In [18] presents an off-line framework to develop performance analysis and post-mortem analysis about the causes of Service Level Objective (SLO) violations. They propose to use TANs (Tree Augmented Naive Bayesian Networks), a simplified version of Bayesian Networks (BN), to determine which resources are most correlated with the performance behavior.

In [17], linear regression is used to build an analytic model for capacity planning of multi-tier applications. They show how linear regression offers successful results for capacity planning and resource provisioning, even under variable workloads.

In [16], authors present an on-line framework that allows to determine if the system is suffering an anomaly, workload change or software change. The authors use linear regression. The idea is to divide the sequence of recorded data into several segments. A segment is divided in two when no single linear regression model gives acceptable error (3%) on the whole segment. If on the two resulting segments there is some model with acceptable error, they determine that we are in front of a software update. If for a performance period it is impossible to obtain a sequence of linear regressions with acceptable errors, the conclusion is that the system is suffering some type of anomaly during that period.

Furthermore, concerning failure and critical event prediction, several studies have been conducted in other areas such as Telecommunications [13, 14, 15]. However, these prediction techniques are not sufficient to manage the complex and varied system health data in computer systems.

3. OUR MODELLING ASSUMPTIONS AND PREDICTION STRATEGY

In this section we describe the mathematical model and assumptions on which our prediction strategy is based. We assume a set of predefined measurements \( M_1, M_2, \ldots, M_n \) of a system. These measurements are taken at different points in time: \( M_t \) is the value of \( M_i \) seen at time \( t \).

Systems are, in theory, deterministic, so the values and evolution of measurements are completely determined by the incoming workload and of the system’s internal state. But,
in practice, the relation between workload, internal state, and measurements is so horrendously complex that we can only think of $M_{i,t}$ as random variables, i.e., as values coming from some (unknown) probability distribution, and deal with them statistically.

The main assumption on our measures, which is also the basis of our prediction strategy, is the following: in a stable, faultless system, with a constant workload, the average of measurements $M_{i,t}$ should remain constant over time, except for random fluctuations (noise) of zero average, that tend to cancel out. We will monitor significant drifts in $M_{i,t}$ over $t$ that cannot be explained by changes in workload as an indication that some fault is occurring; if the drift continues, it is likely to cause some performance degradation and, eventually, a crash.

As an example, suppose that an application uses a minimum of 200Mb, plus 1Mb for each request it is processing. If the workload consists of 50 simultaneous requests, the application uses $200+1\times50=250$Mb. If it later increases to 200 simultaneous requests, we would expect memory use to go up to $200+1\times200=400$Mb. The hypothesis implies that if the load goes down again to 50 requests, the system gradually returns to using 250Mb, i.e., no memory is lost on the way. Similar examples could be given for other measurements, e.g., with the number of threads or processes, % of CPU or bandwidth used, average connection time, number of database accesses per minute, etc. We are especially interested in measures that describe usage of some underlying resource, so that a very high or very low measurement value is problematic.

Now, we consider failures occurring for two reasons:

1) The incoming workload is too large to be dealt with available resources. This should mean that some measurement $M_i$ falls, or evolves to, some unfeasible range. Such failures can, in theory, be anticipated by estimating the resources required by the current workload and predicting that they fall outside the feasible range.

2) The given workload is constant and can, in principle, be handled by the infrastructure, but the system degrades over time. By monitoring the speed at which $M_{i,t}$ varies, at every time $t$ we can estimate a a time $T_{fail} > t$ such that $M_i$ is likely to fall out of the feasible region by time $T_{fail}$: i.e., failure will occur. In particular, let $s_i$ be the speed at which we observe $M_i$ is evolving, and $M_{fail}$ some value such that a crash is likely to occur when $M_i$ reaches $M_{fail}$. Since we have

$$M_{i,T} \simeq M_{i,t} + (T - t) \cdot s_i,$$

and we have $M_{i,T} \simeq M_{fail}$ for $T = T_{fail}$, we can estimate the failure time $T_{fail}$ as:

$$T_{fail} \simeq t + \frac{M_{fail} - M_{i,t}}{s_i}$$

This is the basic equation that underlies our prediction strategy. In other words, our system will say “if measurement $M_i$ continues to drift at the current speed, we expect failure to occur by time approximately $T_{fail}$”.

In order to estimate measurements and their speed in a noise-tolerant way, we should use one of several techniques to smooth them out over some period of time. A common one is to take averages over a window of pre-specified length from the past. We used instead the EWMA (Exponentially Weighted Moving Average) method described e.g., in [26], which assigns more weight to more recent points. On the other hand, values $M_{fail}$ are unknown in general. The role of the machine learning techniques is to estimate such unknown values from past data, but also to account for weaknesses of this very simple model. In particular, they combine the contributions/significance of different measures to the crash, and model less linear behaviors that occur especially near the crashes.

4. THE FRAMEWORK

We have developed a new framework designed towards the ultimate target of our research, that is, an on-line and real-time system for predicting time until failure and approaching crashes. We are, however, at a preliminarily stage testing our method and framework in off-line environment. As long as the instrumentation and prediction methods we use are computationally light, we do not foresee major obstacles in transporting our prototype to the on-line setting.

We can divide our framework in two phases: the training phase and the testing or prediction phase. The training phase has the responsibility to generate the model or classifier. The testing or prediction phase applies this model to the current real-time observations to determine whether the system is approaching a crash. In figure 1, we present an overview diagram of the framework. While this figure presents the whole system, the Recovery Manager is out of scope of this paper because it will be too closely dependent on the system where it is deployed. We will focus on the dark gray modules, the ones that collect and process the information to be handled by the Recovery Manager.

![Figure 1: Prediction Framework](image)

The Monitoring Agent has the responsibility to collect system metrics from the system. We have decided to collect a small set of metrics, easily obtained from any O.S.: throughput, response time, workload, system load, disc usage, swap used, number of processes, number of threads, free system memory, memory occupied by tomcat, number of http connections received and number of connections to the data base. To collect all of this information we have modified Nagios [19], a well-known and used by the industry monitoring tool that allows to execute monitoring gadgets at the system under monitoring every n seconds (in our case will be 15 seconds) and store the results in a file or database.

The typical usage in this offline mode would be the following: We collect all data from several system executions, possibly under different workloads, using the monitoring tool. After that, with the aid of an expert, we locate where in the data set crashes have occurred (for example, which values of response time or throughput are unacceptable). The new data set is used as input for the Enriching Process. The Enriching Process main task is to add additional, derived,
variables to the tuples of system observations. In particular, in the current version of this module, the most important derived variables we introduce are the EWMA-smoothed out values of all measured resources and of their evolution along time. The idea is to be able to detect potential changes of the consumption of any resource. Most importantly, for training purposes, the Enriching Process adds an extra value with the time-to-failure for each tuple of observations; note that this requires looking forward in the dataset to find the next tuple which the expert indicated as crash.

After enriching the data, it is used to build a model, and this model is tested with new data sets or using different data mining techniques like percent split [23]. The flexibility of our framework allows to add new machine learning algorithms for model building easily, or even to have several algorithms and create a committee of models for increasing the accuracy of the predictions. This latter idea was used, e.g., in [24], where they have four models and choose the result by general consent (at least 3 out of 4 must agree). Figure 2 presents the activity diagram between the different modules presented before.

5. TEST EXPERIMENTS

5.1 Experimental Setup

In our experiments, we have used a multi-tier e-commerce site that simulates an on-line book store, following the standard configuration of TPC-W benchmark [20]. We have used the Java version developed using servlets and using Mysql [21] as database server. As application server we have used Apache Tomcat [22]. TPC-W allows us to run different experiments using different parameters and under a controlled environment. These capabilities allow us to conduct the evaluation of our approach to predict the time until failure. Table 1 details the machines used.

TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an on-line book store) in sessions. A session is a sequence of logically connected (from the EB point of view) requests. Between two consecutive requests from the same EB, TPC-W computes a thinking time, representing the time between the user receiving a web page s/he requested and deciding the next request. In our experiments we have used the default configuration of TPC-W. Moreover, following the TPC-W specification, the number of concurrent EBs is kept constant during the experiment. To inject transient software errors which consume randomly resources we have modified a servlet of the TPC-W implementation. This servlet is able to inject two type of transient errors: Memory and Threads consumption, or even both of them at the same time. For memory errors, this servlet computes a random number between 0 and N. This number determines how many requests use the servlet before the next memory leak is injected. The memory consumption injected in our experiments is fixed (1MB) in all experiments. We could have made this size random too. But, again, the average consumption rate would depend on the average of this random variable, with fluctuations that become less relevant when averaged over time. For threads starvation, the servlet computes two random numbers: The time until next creation is a random number between 0 and M and the number of threads to create is also a random number between 0 and K. The variation of memory consumption depends on the number of clients and the frequency of servlet visits. According to the TPC-W specification, this frequency depends on the workload chosen. TPC-W has three types of workload (browsing, Shopping and Ordering). In our case, we have conducted all of our experiments using shopping distribution. The EBs in this workload visit the servlet modified around 20%. This makes that with high workload our servlet injects quickly memory leaks, however with low workload, the consumption is lower too. In the threads error case, the number of threads doesn’t have any relationship with the frequency of the servlet visits, making completely independent the error frequency than the workload.

5.2 Experimental Results

5.2.1 Model Evaluation under Fixed Environments

In our previous work [**], we evaluated two families of algorithms: linear regression and decision trees (RepTree and M5P), and we concluded that decision trees are in general better options because linear regression is extremely sensitive to outliers. Outliers are very common when the system is suffering transient errors, and so linear regression obtained poor results. We further identified the M5P algorithm, which combines linear regressions with decision tree, as superior to using either technique alone. Also we measured the accuracy using the Mean Average Error (MAE) over all the predictions. However MAE is not as important as having low error near the crash, the point where we want to be the most accurate as possible, so we re-evaluated the method focusing on the last minutes before the crash (we chose the last 10 minutes as critical period, because it is
time enough to take any recovery or preventive action to avoid the crash).

To evaluated the effectiveness of our approach we have conducted three experiments: a) injecting Threads, b) injecting memory leaks and c) injecting a mix of both type of errors. The first error is completely independent of the workload, although it is randomly with two grades of freedom: the time until next injection \((M = 60)\) and the number of threads created \((K = 30)\). To simulate a random memory leak, we injected a memory leak of 1MB on \(N/2 = 15\) visits to the faulty servlet in average, with constant workload (recall that we fixed \(N = 30\)), and Tomcat’s heap size set to 1GB. Finally, to mix both type of errors, we injected both errors using the same variables defined before.

For the first two environments, the model was trained using different workloads (number of EB’s) all under these error schemes. We merged the five logs resulting of using 5, 25, 50, 100, and 200 Emulated Browsers (EBs) into a single training data set. After that, we evaluated the model under two different workloads never used to the training process, 75 and 150EB’s but, this time, unlike [[*]], we focused on the accuracy during the last 10 minutes before the crash. For the third experiment, we merged the training files from both experiments before to obtain the new training file, so the model never was trained with any experiment where the both errors where injected simultaneously, so the model had to deduce the time until crash under both errors from logs where only one was injected at time.

In tables 2 and 3, we can observe the MAE and standard deviation \((S)\) comparison between three algorithms under the two first errors: Threads injection and memory injection. In table 2 we can observe how all three algorithms obtain a very good results with a low errors, in fact under 49.60 seconds. Table 3 presents the results of the complete experiment are more or less the same (M5P continues being the best option), but if we focus on the last 10 minutes we can observe how only M5P is able to adapt itself to the new environment obtaining very promising results. Again, we can observe how Linear Regression is useful to predict the time until crash in the last 10 minutes of experiment.

Finally, table 4 presents the error obtained by every algorithm when we run an experiment where we mixed the both type of errors. We remember in this experiment the model was trained only with logs from executions (with different workloads) where only one error was injected, to evaluate the adaptability of the algorithms to complete new circumstances. We can observe clearly how the results of the complete experiment are more or less the same (M5P continues being the best option), but if we focus on the last 10 minutes we can observe how only M5P is able to adapt itself to the new environment obtaining very promising results. Again, we can observe how Linear Regression when we run an experiment where we mixed the both type of errors.

<table>
<thead>
<tr>
<th>Table 2: MAE - S for &quot;Threads injection&quot; Experiment for 75EBs and 150EBs. (Values are in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5P</td>
</tr>
<tr>
<td>All 75EBs</td>
</tr>
<tr>
<td>Last 10 min</td>
</tr>
<tr>
<td>75EBs</td>
</tr>
<tr>
<td>All 150EBs</td>
</tr>
<tr>
<td>Last 10 min</td>
</tr>
<tr>
<td>150EBs</td>
</tr>
</tbody>
</table>

In tables 2 and 3, we can observe the MAE and standard deviation \((S)\) comparison between three algorithms under the two first errors: Threads injection and memory injection. In table 2 we can observe how all three algorithms obtain a very good results with a low errors, in fact under an independent error of the workload all algorithms are useful to predict the time, even Linear regression obtains great results. The reason is because the random error injected, in average injects \(K/2 = 15\) threads every \(M/2 = 30\) seconds, becoming in quite stable and predictable behavior.

On the other hand, in table 3, we present the MAE under a dependent error of the workload and the frequency of the use of the servlet modified. In this case, the results are worse but promising. In this case, the decision tree algorithms (M5P and RepTree) obtain better results than Linear regression for the complete experiment, however if we focus on the last 10 minutes of experiment M5P and Linear Regression obtain better results. But the Linear Regression results are debatable, because Linear regression starts to predict 0 seconds to crash too early (more than 30 minutes before the crash) becoming useless. For this reason, M5P becomes more effective to predict the time until crash in the last 10 minutes of experiment.

<table>
<thead>
<tr>
<th>Table 3: MAE - S for &quot;Memory injection&quot; Experiment for 75EBs and 150EBs. (Values are in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5P</td>
</tr>
<tr>
<td>All 75EBs</td>
</tr>
<tr>
<td>Last 10 min</td>
</tr>
<tr>
<td>75EBs</td>
</tr>
<tr>
<td>All 150EBs</td>
</tr>
<tr>
<td>Last 10 min</td>
</tr>
<tr>
<td>150EBs</td>
</tr>
</tbody>
</table>

5.2.2 Detecting Imminent Crashes under a Fixed Environment

<table>
<thead>
<tr>
<th>Table 4: MAE - S for &quot;Mix injection&quot; Experiment for 75EBs and 150EBs. (Values are in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5P</td>
</tr>
<tr>
<td>All 75EBs</td>
</tr>
<tr>
<td>Last 10 min</td>
</tr>
<tr>
<td>75EBs</td>
</tr>
<tr>
<td>All 150EBs</td>
</tr>
<tr>
<td>Last 10 min</td>
</tr>
<tr>
<td>150EBs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5: Confusion Matrix using J48. X/Y, with X for 75EBs and Y for 150EBs for Mix Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>Green</td>
</tr>
<tr>
<td>Predicted</td>
</tr>
<tr>
<td>Values</td>
</tr>
</tbody>
</table>
After this focused evaluation, we concluded that the best strategy to predict the time until failure would be to have different models depending on the state of the system. M5P in average seems the best option because obtains the best results in quiet every comparison, however its results are not too much good when the crash is far. Because of this, and in itself, it would be interesting to simply distinguish whether or not a crash is approaching. This called for classifier algorithms, whose output is not a number but one among a binary, ternary, etc. set. To allow for some flexibility, we distinguished three rather than two periods: Tuples in the dataset were labeled Red, Orange, or Green. Tuples in the last 10 minutes before the crash were labeled Red, those in the 10 minutes before the Red zone were labeled Orange, and all others (i.e., at least 20 minutes before the crash) were labeled Green. We still are primarily interested in distinguishing Red from non-Red, but we will not count Red tuples classified as orange as very severe mistakes. Red tuples classified as Green are probably the most expensive mistake, as they mean an unpredicted crash. Green tuples classified as Red are false positives: they could mean starting preventive measures before they are strictly necessary or even without being necessary at all, which may be a nuisance but not as much as a crash.

Among the many classifiers in the literature, we took three incorporated into the WEKA package [23]: J48, an implementation of the well-known C4.5 decision tree inducer, NaiveBayes, a probabilistic classifier based on Bayes’ rule, and IBk, the nearest neighbor classifier.

### Table 6: Confusion Matrix using NaiveBayes. X/Y, with X for 75EBs and Y for 150EBs for Mix Injection

<table>
<thead>
<tr>
<th></th>
<th>Green</th>
<th>Real Value</th>
<th>Orange</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>179/47</td>
<td>0/0</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>Predicted Orange</td>
<td>194/248</td>
<td>14/0</td>
<td>15/0</td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>Red</td>
<td>7/19</td>
<td>3/12</td>
<td>3/3</td>
</tr>
</tbody>
</table>

We used WEKA’s default options in all three cases. We remark our use of off-the-shelf classifiers to indicate that we included hardly no domain knowledge into the prediction system and that, obviously, there is ample room for improvement by designing ad-hoc algorithms. The reason for choosing these three is that they are computationally light (with Naive Bayes being the cheapest and IBk the most expensive of the three). The results are shown in the form of confusion matrices, indicating how many examples of each class (red, orange, green) are classified as in each class. These matrices provide better information than simply the number of errors. Perfect predictions are those in the diagonal. E.g., red-as-orange confusions are wrong, but intuitively not as wrong as Red-as-green. Due to limited space we only present the results obtained in the third experiment, where we mix the both type of errors without an specific training for this scenario.

Table 5 presents the results obtained using J48 under the mixed experiment. We can observe that J48 classifies many Greens as Oranges but makes none or very few “severe” Green-Red mistakes.

Table 6 presents the results obtained using Naive Bayes under the mixed experiment. We can observe that Naive Bayes model also makes few Green-Red mistakes, but tends to place a good number of both Reds and Greens in the Orange zone. Although, it has problems to predict the red zone with 75EBs as a workload.

Table 7 presents the results obtained using IBk under the two data sets in the Mixed experiment. We can observe that IBk model has an acceptable accuracy for all zones, however has mistakes in all of them. In particular, they do classify 67 Green tuples as Red (a severe mistake which could provoke false positives) in the all datasets.

The next step, not yet carried out, would be to combine the different predictors (numerical and classifier) in a wise advisory board to predict the crash of the system and the time until failure with acceptable level of accuracy. In particular, different classifiers would be trusted in different types of system states.

#### 5.2.3 Model Evaluation under a Dynamic Environment

To create a dynamic environment we divided the experiment en 4 phases: the first phase we don’t inject any error, after that phase (30 minutes) we start to inject at determinated ratio the error, after 30 minutes later, the ratio is changed, and after other 30 minutes the ratio is changed again and we wait until crash. The idea is to test our predictor’s ability to detect the different rates and answer appropriately. That

![Figure 3: Variation of Tomcat memory used during the experiments](image-url)
In the threads experiment the values of the variables are varying as follows: The first phase obviously we inject 0 threads (phase 0), after that $M = 90$ and $K = 30$ (phase 1), 30 minutes later, we change the value of the variables to $M = 120$ and $K = 15$, and finally, $M = 60$ and $K = 45$ until crash the server. In the case of memory error experiment the $N$ variable varies from 0 to 30 ($N = 30$, after 30 minutes change to $N = 15$ and finally to $N = 75$. As we can see, in the first experiment we start with a medium injection rate, after that we reduce the injection rate and finally, we increase aggressively the injection rate. However, in the second scenario, we change the rate variation from medium to aggressive and the experiment ends with a slow rate of injection. In figure 3, we present memory resource varies during the experiment in front of a fixed $N = 30$ scenario.

In tables 8 and 9 we present the MAE and S of the two first experiments. We can observe how M5P is the best option (lower error) to predict the time until crash for memory injection experiment, however RepTree is better when the error is independent from the workload like Threads injection Experiment. Although, if we evaluate the last 10 minutes of every experiment, we can conclude that RepTree obtains better average results, though all algorithms obtain debatable results in the last 10 minutes, becoming useless to predict the time when the crash is imminent. Again, if we focus on the last 10 minutes of 150EBs under Threads injection (table 8, Linear Regression obtains the best numbers, however this value is not true because Linear Regression starts to predict 0 seconds for crash far from that (more than 30 minutes before). On the other hand, in these dynamic experiments we wanted to evaluate the adaptability of the algorithms under the change of the rhythm of the resource consumption. For example, in figure 4, we present the time predicted by RepTree in front of the real time until crash in the mixed error experiment. We can observe during the first 30 minutes we don’t inject any error, under these circumstances the model was trained to predict a very big number (3 hours until crash), like an infinite value, and we can observe how the model tries to achieve this value, after that we start to inject errors (both memory and threads) and we can observe how the model starts to reduce the time until crash, but more slow than the real time, but a very interesting result happens after that. We started to inject memory leaks aggressively and threads slowly. Due to these errors, the real time until crash will be nearer and we can observe how immediately, RepTree is able to detect the change on the rhythm of the resource consumption and it adapts the time predicted quickly. The same happens when the time until crash changes again. This is a very promising behavior to detect changes and avoid to apply recovery actions before they are really necessary.

![Figure 4: Predicted vs. real time until crash using RepTree on Memory and Threads Consumption](image)

### 5.2.4 Detecting Imminent Crashes in a Dynamic Environment

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Orange</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>128</td>
<td>35</td>
</tr>
<tr>
<td>Green</td>
<td>241</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 11: Confusion Matrix using J48 for 100EBs</th>
<th>Green</th>
<th>Real Value</th>
<th>Orange</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>Red</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
Finally, we considered whether, under a changing workload, one can distinguish Red from Green zones. Again, we used the J48, Naive Bayes and IBk classifiers. We only present the results obtained by J48 in Table 11 for the experiment where we injected the both errors simultaneously. We present only J48 due to it obtains the best results, because Naive Bayes and IBk classified several red instances like green, provoking false negatives and becoming useless to distinguish Red from Green zones. We can observe that J48 is able to determine the alarm zone (Red and Orange zones) clearly. An interesting point is the fact that the 128 instances from Green zone are placed in Orange zone, are the instances just before the right Orange zone (the frontier). On the other hand, other interesting result that we could extract from our experiments where the Naive Bayes and IBk were better to predict when the system is in the Green zone. This behavior invites us to propose as a future work combining several models to increase the prediction accuracy.

6. CONCLUSIONS AND FUTURE WORK

We have shown that, for two particular type of transient errors it is possible to predict with reasonable accuracy the time remaining for a crash and whether or not a crash is imminent. This opens the possibility of combining prediction with self-healing techniques to avoid crashes or malfunctions, which can be extremely expensive or damaging. Among the main lines for future work are: Turning our off-line system into a truly online, real-time one. Trying a wider variety of prediction and classification methods, and possibly developing ad-hoc ones, for improved accuracy. Combining several models (obtained by one, or several, learning algorithms) also to improve accuracy. Finally, testing our approach on a real system, and combining it with actual self-healing techniques (i.e., implementing the Action Recovery Manager module from Figure 2), like [27]. Also, we want to measure in detail the impact on the system under monitoring performance to evaluate the effectiveness of our approach.

7. REFERENCES


